AFRL MSRC C Shell (csh) Programming Tutorial

Topic List

[This is intended to be a tutorial on scripting in the C shell csh. The interested reader is referred to Anderson, *The Unix C Shell Field Guide*, and similar texts.]

C Shell (csh) Programming

Note: '␣' denotes a space (' ') character.

C shell has similar structures to the C programming language.

The C shell csh and its derivatives are popular shells that have syntax similar to the C programming language.

The following are two examples of a simple shell script. As the reader can see, this is a series of commands to change to the user's $HOME directory, create a subdirectory, and print out the line from the env command containing the string PATH; the reader could execute these same series of commands from the command line.

Basic shell script

```
#!/bin/csh
cd $HOME
mkdir foo
env | grep PATH
```

or

```
#!/bin/csh
cd $HOME ; mkdir foo
env | \
    grep PATH
```

Note: commands can be separated by a newline character or ';'. '\' is used for line continuations.

The first line in the file "#!/bin/csh" tells the shell what interpreter to use with the script as input. Due to space considerations, the "#!/bin/csh" line will be omitted in the examples as they are code snippets. The second example demonstrates the use of ";" to combine commands on the same line and "\" to split command lines on multiple lines.

Any text following a "#" character is treated as a comment.

These scripts must have the execute-bit enabled (chmod u+x filename) to execute these scripts as commands.

The utilities expr and find will be used in the examples. These utilities have individual man pages.

> The following are a few utility commands that will be used in subsequent examples.
>
> expr – integer arithmetic operations
> find – locates files with specific attributes

As with other programming languages, variables provide a means of storing and recalling data via a naming scheme.

> Variables
>
> *name* must begin with a letter or '_' character, followed by zero or more letters, numbers, and '_' characters.
>
> Assignment: set *name* = *value*
> set *name*[**n**] = *value*
> set *name* = (*value-1* ... *value-N*)
>
> Environment variable: setenv *name value*
>
> Value: $*name*
> $*name*[**index**]
>
> Replaces with value: $*name*, "$*name*", \`$*name*\`
> No change: \$*name*, '$*name*'

The three variations of the set command assign to a shell variable, the n^{th} element of a list variable, and a list variable respectively. These variables are only defined in the current shell process.

setenv assigns values to environment variables, which are passed to subshells or child shells.

The value of a shell or environment variable is referenced by prepending a "$" to the name. Note that using double quotes (") or gravé quotes (\`) expands to the variable's value whereas escaping the "$" or using single quotes (') does not. Due to variable modifiers (not discussed), enclosing the name of the variable in braces ({}) avoids accidental interpretations when combining variable values and ":"'s.

The example illustrates the various methods of assigning and using shell variables, with the output following. The pwd command outputs the full path of the current directory. The echo command prints its arguments to standard output. The env command prints out the names and values of the environment variables in the current shell process. The grep command prints out occurrences of its first argument in its input (standard input in the example).

Example: comparing shell and environment variables

```
pwd

set foo = ”Value of foo”
setenv AKIRA ”pwd”

echo $foo “$AKIRA”
echo \$foo ’$AKIRA’
echo ‘$AKIRA‘

env | grep AKIRA
env | grep foo
```

Result:

```
/home/user
Value of foo pwd
$foo $AKIRA
/home/user
AKIRA=pwd
<blank line>
```

The script: 1) outputs the full path of the current directory, 2) assigns values to the shell variables **foo** and **AKIRA**, 3) makes **AKIRA** an environment variable which is available to child processes, 4) outputs the various forms of the two shell variables, and 5) outputs matches of the two shell variables in the enviroment variable output.

The first line of the output is the full path of the current directory.

The second line of the output is the concatenation of the values of the two shell variables. This shows that both the unquoted form and quoting with double quotes ("") expands to the value of the shell variable.

The third line of output are the variable names preceded by a '$' character. Escaping (preceding by '\') the '$' or enclosing in single quotes (´´) prevents expansion to the value of the shell variable. This form is useful if '$' does not precede a valid shell variable name, *e.g.*, currency expressions. Delayed expansion is also useful when the shell variable has no value in the current shell process (*e.g.*, writing out a shell script, executing a command on a remote system).

The fourth line of output is the full path of the current directory. The gravé quotes (``) execute the value of the shell variable as a command in a subshell. Here, **$AKIRA** expands to **pwd**, and the output of the command is stored as a string.

The last two lines of output (the last line is blank) show that **AKIRA** was defined as an environment variable whereas **foo** was not.

The following is a list of common shell variables. Some of these will be used in later examples.

Useful variable values

$PATH	colon-separated list of directories in which to search for commands
$LD_LIBRARY_PATH	colon-separated list of directories in which to search for shared objects
$MANPATH	colon-separated list of directories in which to search for man pages
$HOME	user's home directory
$USER	login name
$PWD	full path of current directory
$status	exit status value of last command executed
$$	process id (PID) of the current process
$0	name of called script
$1–$9 $argv[1]–$argv[9]	up to the first nine positional parameters
$#argv	count of positional parameters (can be larger than nine)
$*	string of all the positional parameters

Note that **argv** is a variable list. The range of indices is 1 to **$#argv**.

Constructing strings

Assume that the value **$foo** is the string “JIRO,” the value **$bar** is the string “mohmoh-rehnjah,” and the value **$null** is the empty string (“”) in the following examples.

Expression: $foo5
Equivalent: “” [undefined variable]

Expression: ${foo}5
Equivalent: “JIRO5”

Expression: x$bar
Equivalent: “xmohmoh-renjah”

Expression: ´x$null´
Equivalent: “x$null” [no expansion]

Expression: $bar´ has explosive jewelry.´
“$bar has explosive jewelry.”
Equivalent: “mohmoh-renjah has explosive jewelry.”

Expression: \`echo ${foo}“5$null”\`
Equivalent: “JIRO5” [result of echo evaluation]

These examples demonstrate constructions of strings which combine literal strings and shell variable values. Except in the case of characters that have special meaning in the shell (*e.g.*, spaces, dollar signs), quoting in strings is not necessary. However, quoting does avoid errors with strings used as operands.

The first two examples demonstrate the wrong and correct methods of concatenating a shell variable value and an alphanumeric string. Since **foo5** is a valid variable name, **$foo5** is syntactically correct even though the variable has no value. By surrounding the name with braces, the shell interpreter expands the value **$foo**, and concatenates the string “5” to it.

In the third example, the value **$bar** is concatenated to the string “x.” In this case, the braces are not needed.

The fourth example shows that single quotes prevent expansion of shell variable values.

The fifth example shows equivalent combinations of shell variable values and literal strings.

Although the sixth example is contrived, it shows that any of the constructed strings are valid arguments to commands

Logical expressions and file inquiry

The if and while constructs use the results of conditional expressions or integer values to make decisions.

The logical operators are similar to their C counterparts. In addition, the file inquiry operators determine properties of their file arguments.

As in the C programming language, not-zero is true, and 0 is false. This convention is the opposite of the Unix return values 0 (success/true) and not-zero (failure/false).

File inquiry and logical operators	
-f *value*	true if *value* is an ordinary file
-d *value*	true if *value* is a directory
-z *value*	true if file *value* is empty
-r *value*	true if $USER has read-permission on file *value*
-w *value*	true if $USER has write-permission on file *value*
-x *value*	true if $USER has execute-permission on file *value*
==, !=	string comparison operators
==, !=, <, <=, >, >=	integer comparison operators
!	negation operator
&&, ||	logical AND, OR
(,)	group expressions to increase precedence

Examples:

1. “$LIBPATH” == “”
2. -x /bar/RECORD.USAGE
3. “$LD_LIBRARY_PATH” != “”
4. $# > 0
5. ! -d ./foo

Explanation:

1. true if **$LIBPATH** is the empty string (“”)
2. true if **/bar/RECORD.USAGE** is an executable file
3. true if **$LD_LIBRARY_PATH** is not the empty string (“”)
4. true if the number of command-line arguments is greater than zero (0)
5. true if ./foo is not a directory

Making choices (if and switch):

if (*expression*) *command*

if (*expression*) then
    command-1
    [⋮
    command-N]
[else if (*expression*) then
    command-1
    [⋮
    command-N]]
[else
    command-1
    [⋮
    command-N]]
endif

The if construct sequentially evaluates each *expression* until 1) one evaluates as true, 2) the else clause is reached, or 3) endif (end of the construct) is reached. For cases 1) and 2), all commands are executed up to the next else if, else, or endif.

In the example, the tty command checks if the standard input stream is connected to a terminal without generating output. The return value **$status** is checked (0 is success/true in Unix, but false in C), and the erase character is set to control-H, or backspace, if the conditional expression is true.

Example:

This sets the erase character to backspace only when connected to a terminal; stty will give an error when the stdin is not a terminal. tty tests stdin.

```
tty -s
if (! $status) then
    stty erase ´^H´
endif
```

This snippet is useful in **.login** file which is executed at the beginning of a login session; the clause prevents execution for non-interactive logins such as batch processing.

```
switch (value)
    case case-a1:
    [ ⋮
    case case-aN: ]
        action-1
        [ ⋮
        action-N ]
        breaksw
    [ ⋮
    case case-N1:
    [ ⋮
    case case-NN: ]
        command-1
        [ ⋮
        command-N ]
        breaksw ]
   [ default:
        command-1
        [ ⋮
        command-N ]
        breaksw ]
endsw
```

Note: *value* is typically the value of a variable ($*name*) or the output of a command (\` *command*\`) where the grave ('\`') turns the output of a command into a string. breaksw denotes the end of the actions for the group of cases.

value is compared to all of the *case-** terms; if a match is found, the commands up to the terminating breaksw are executed. If no match is found, the default *default* is matched, if it exists, and the associated commands are executed.

The command ping gives a simple test of connectivity to a remote machine. The command rsh allows command execution on remote machines if user **$USER** has login access. However, different operating systems have differing full paths to these utilities. The following example is one method to simplify this process for use later in a script.

Example:

Sets shell variables with full pathnames for ping and rsh

```
set OS = `uname -s`

switch ($OS)
    case IRIX*:
        set PING = /usr/etc/ping
        set RSH = /usr/bsd/rsh
        breaksw

    case OSF1*:
    case SunOS*:
        set PING = /usr/sbin/ping
        set RSH = /usr/bin/rsh
        breaksw

    case Linux*:
        set PING = /bin/ping
        set RSH = /usr/bin/rsh
        breaksw

    default:
        echo "Unknown operating system"
        exit 1
        breaksw
endsw
```

The output of the uname command gives the operating system name; this is assigned to the shell variable **OS**. This switch construct selects the values to assign to the shell variables PING and RSH for the operating systems that it recognizes. The value **$OS** is compared to the various case strings; the "*" in all the cases matches any string. The "default" case matches any case not previously matched.

By assigning values to shell variables, the commands that follow this switch construct can use the values of the shell variables as opposed to the actual full pathnames. Thus, the same script can function on multiple machines minimizing maintenance errors.

As with other programming languages, switch constructs can be the body of if constructs, and *vice versa*.

The following snippet first checks whether the value **$CPU** is the empty string.

If the value is the empty string, the output of the hostname command is compared to the cases. Here both the "*", which matches zero or more characters, and the range operator (here denoting a digit) are used in the patterns to match. If a pattern matches, **CPU** is assigned the associated value. If no pattern matches, a message is printed to standard output, and the script exits.

Example:

A method to set the CPU environment variable for known machines

---

```
if ("$CPU" == "") then
    switch (`hostname`)
       case aaa-[0-9]*:
       case bbb-[0-9]*:
          setenv CPU alpha
          breaksw
      case ccc-[0-9]*:
          setenv CPU beta
          breaksw
      case ddd:
          setenv CPU gamma
          breaksw
      default:
          echo `hostname` "does not define CPU environment \
             variable"
         echo "Contact user@host.domain about this \
             problem"
         exit 1
          breaksw
    endsw
endif
```

Positional parameters

At a basic level, the positional parameter values $1 ... $9, $argv, and $*, and their respective count $# are the references to the command-line arguments of the script; $0 is the name of command that invoked the script. The initial nine positional parameter values (where applicable) are referenced by the values $1 ... $9.

If more than nine (9) positional parameters exist, the remaining positional parameters can be moved into the $1 ... $9 values using the shift command with no argument; with a variable list name as an argument, the elements of the variable list are shifted. In either case, the first element is lost, and all elements shift to the left (lower index) by one position.

Alternatively, $argv[k] references the kth argument where k ranges from 1 to $# (or $#argv).

As with other variable lists, the elements of **argv** can be referenced as **$argv**[*num*] where *num* ranges from 1 to **$#** (or **$#argv**).

To illustrate the use of the shift command, the list of positional parameter values is considered to be a space-separated list with the lowest index on the left. The command

shift [**variable-name**]

will shift all the elements in the list **variable-name** (or the positional parameters if no argument is present) by one position to the left, losing the value to the left of the first position. In addition, **$#** is decremented by one as well.

This example demonstrates the passing of command-line arguments to another command. This snippet could be used in a wrapper script where the setup operations or logging might be incorporated in addition to executing the command. As these operations would generally execute in a subshell, the calling shell environment would not be affected.

Example:

Passing the command-line arguments to another command.

```
command=`basename $0` # Gets name of command
/software/$CPU/bin/$command $*
```

The assignment of **command** uses the basename command to get the name of the script as it was called, less any directory path information. The gravé quotes (``) change the output of the command into a string for the assignment.

In a wrapper script, the value **$0** would be the symbolic link associated with the actual script, *e.g.*, the link **snafu** created by

    ln -s wrapper.sh snafu

would start the executable (**/software/$CPU/bin/snafu** when invoked. This allows one script to create a consistent operating environment for multiple related executables.

Looping: foreach, while

foreach *name* (*value-1* ... *value-N*)
    command-1
      ⋮
    command-N
end

For each iteration of the loop, variable *name* is successively assigned *value-1* through *value-N*, and the commands in the body are executed.

---

while (*expression*)
command-1
⋮
command-N
end

The while construct loops over the commands between the while and end as long as *expression* is true.

Example:

This example creates links which associate commands with a wrapper script.

```
cd /base_$CPU

foreach file (`cd /software/$CPU/bin ; ls`)
   if (! -f $file) then
     ln -s /software/scripts/app.sh $file
   endif
end
```

After changing to the directory for the links, the foreach loop iterates over the names of executable files; by changing to the directory before executing the directory listing, only the filenames are output. Note: the commands within the gravé quotes are executed in a subshell, and do not directly affect the current process.

To avoid error messages from ln, an if construct checks that the link does not already exist. The wrapper script (called **/software/scripts/app.sh**) is linked with the name of an executable binary (**$file**).

Example:

Changes the uppercase letters in the names of a group of files to lowercase

```
foreach file (`ls`)
    set new_file = `echo $file | tr ´[A-Z]´ ´[a-z]´`
   if ("$new_file" != "$file") then
       mv $file $new_file
    endif
end
```

$file successively takes values of the filenames in the current directory. **new_file** is assigned the value **$file** with all the uppercase letters changed to their lowercase equivalent; as tr only works on streams, the value **$file** is echo'ed and piped to tr.

The values **$new_file** and **$file** are compared. If they are not equal, the file **$file** is renamed **$new_file**. Both **$file** and **$new_file** are quoted as a precaution against empty-string values.

This example demonstrates the while loop which terminates when its condition returns false (not 0).

Example:

Iteratively operating on positional parameters

```
while ($# > 0)
    echo $1
    shift
end
```

$# is the number of positional parameters, so the loop continues as long as it is non-zero. For each loop iteration, the first parameter (**$1**) is output, and each parameter is moved one position down with **$1** being discarded. **$#** is updated with each shift command.

Although this loop only prints out each parameter, the action could be more involved such as moving files or processing terms.

The outer loop continues until the value **$STATE** is 0 (success in Unix).

Example:

Getting a successful file transfer

```
set STATE = -1
while ($STATE != 0)
   rcp foo.dat host:/home/user
   set STATE = $status
  if ($STATE != 0) then
     kinit -l 10h
    while ($status != 0)
       kinit -l 10h
     end
   endif
end
```

The outer loop continues until the value **$STATE** is 0. Note: the value **$STATE** is initialized to be not 0.

The body of the outer loop: 1) tries to copy a file to a remote host, 2) assigns the return value of the copy command to **STATE**, and 3) performs additional processing if the copy failed (value **$STATE** is not 0).

The assumption to continue processing is that the remote copy command **rcp** should only fail in a recoverable way: if there are no valid credentials (loss of network connectivity is not a recoverable error). Thus, when a failure occurs, the loop executes until the credentials have been established.

Note: the expressions "$*name* != 0" could also be written "$*name*" as not-zero is "true."

eval command

When performing the same operation on a collection of variables, it is convenient to use the looping structures that work when the value of a variable is allowed to change. The eval command evaluates its argument, then executes the expanded string as a command.

Example: sets the value of the variable to its name

```
foreach match (rusage select test)
    eval ´set ´$match´ = $match´
end
```

This apparently trivial example demonstrates the utility of the eval command. The value of **$match** is successively set to three values. On the left-hand-side of the assignment, the unquoted string expands to the value **$match**. On the right-hand-side of the assignment, the single quotes prevent expansion of **$match**. So, before applying eval, the argument would look like (for **$match** being rusage):

```
´set ´rusage´ = $match´
```

which is clearly an illegal form. eval removes one level of quoting, and evaluates the result. Thus the command to evaluate would be:

```
set rusage = $match
```

From this trivial example, other sources might be used to assign **$match**. As this is the variable name to be assigned, a similar eval statement would assign a changing variable name to a string.

The variation

```
eval ´set ´${match}1´ = $´$match
```

would expand to the command

```
set rusage1 = $rusage
```

allowing one to assign other variables via the loop.

Doing math

For integer math, there are built in commands as well as the expr command. Note: the multiplication operator is *, not *, as the latter is special to all shells. expr has additional operators besides the usual math operators.

The expr command takes as its arguments the expression to be evaluated where the operands and operators are space-separated. Although other operators are noted in the man pages, for this discussion, only the mathematical operators +, -, *, /, and % are considered.

This example tests the scalability of an MPI program. The program **foo** is assumed to be an MPI executable that can be started by the command mpirun.

Example: scaling test

```
set count = 1
while ($count <= 128)
    mpirun -np $count foo
    set count = `expr $count \* 2`
end
```

The variable **count** stores the processor count, and begins at 1. The loop continues while the value **$count** is less than or equal to 128.

For each loop iteration, **foo** is run on **$count** processors, then **count** is assigned the value of twice **$count**. As seen in previous examples, the gravé quotes (`) change a command's output into a string, and do not prevent expansion of shell variables.

Built-in math

The format

@ ⊔ *variable-name* ⊔ *operator* ⊔ *expression*

where spaces between the '@' and *variable-name*, and between *variable-name* and *operator* are required; spaces are recommended for clarity. *operator* can be the usual C assignment operators (=, +=, -=, *=, /=) as well as post-increment/decrement (++, −−).

expression can be a variable value or integer expression.

Example: scaling test (built-in math)

```
@ count = 1

while ($count <= 128)
    mpirun -np $count foo
    @ count *= 2
end
```

continue, break

It is convenient to avoid executing all the commands in a loop if some condition is met. continue stops the current iteration of a loop and moves to the next one. break terminates the current loop.

Example:

Changes the uppercase letters in the names of a group of files to lowercase (using continue)

```
foreach file (`ls`)
    set new_file = `echo $file | tr '[A-Z]' '[a-z]'`
   if ("$new_file" == "$file") then
       continue
    endif
   mv $file $new_file
end
```

$file successively takes values of the filenames in the current directory. **new_file** is assigned the value of **$file** with all the uppercase letters changed to their lowercase equivalent; as tr only works on streams, the value of **$file** is echo'ed and piped to tr.

The values **$file** and **$new_file** are compared. If they match, no change is necessary, and the continue statement returns control to the top of the loop. Otherwise, the filename changes to **$new_file**. This is a rewriting of a previous example to illustrate the use of the continue statement.

find command

The find command is useful for locating files with specific attributes. By default, find generates a list of files that match the search criterion. The -exec option specifies a command to execute where "{}" denotes the current match; the argument to -exec must be terminated by "\;".

The reader should reference the man pages for find.

Useful find options:

-name *filename-specification*	finds filenames matching the specification
-perm *permission-specification*	finds files with matching permissions
-ctime *number-of-days* -mtime *number-of-days* -atime *number-of-days*	finds files with creation, modification, access times in *number-of-days* days
-type *type-specification*	finds files of the specified type
-exec *command* \;	executes *command* with "{}" (the current match) as an argument.
! *option*	negates its argument
\(, \)	groups options
option-1 -a *option-2*	both specifications must be true for match
option-1 -o *option-2*	either specification must be true for match

```
find . -name ´*.c´     # Finds C source files

# Changes permissions to allow group/world read access
find /software/ -perm 0600 -exec chmod go+r {} \;
```

The first example locates files that end in '.c.' As '*' is special to the shell, it is quoted so that the wildcard is not expanded, but passed to find as a parameter.

The second example finds files in the directory **/software** that have only user read-write permissions. The identified files then have group and other read permissions added. In the chmod command associated with the **-exec** option, {} refers to the found file, and \; indicates the end of the command.

In-line input redirection or here documents

Although the stdin for a command can be redirected from a file, it is useful in a script to have the data in the script as opposed to depending on a file. "Here" documents provide this functionality.

---

command <<*flag*
line-1
⋮
line-N
flag

All the lines up to the line containing only *flag* are through stdin for *command*. Note: no spaces between "<<" and *flag*. All shell variables are expanded by default unless *flag* is preceded by a "\".

Instead of reading from standard input, this example uses a "here" document for input. An advantage to this is that it is does not require the user to interactively enter the message, and allows a combination of static data with dynamic content (this could be done in a loop over valid users).

Any command that reads from the standard input stream can potentially have a "here" document as an alternative to an input file.

Example: mails notice to user

```
if (! -f $HOME/.login) then
    mail -s "Missing $HOME/.login" $USER <<EOF
Dear user $USER:
   Alert: the file $HOME/.login is missing.
   System administrator
EOF
```

This example checks for a login startup file, and mails a note to the user if the file does not exist.

source command

The source command executes its argument, a valid shell script, in the current shell. Thus, unlike executing the shell script by itself, it effects changes in the current shell as opposed to running in a child shell. The source command is useful for modifying the current process's environment or adding aliases.

exit command

exit
exit(*expression*)

The exit command terminates the current shell script at the point at which it is executed rather than allowing it to complete. If an optional argument *expression* is included, it returns an exit status of the value of *expression*; otherwise, it returns the exit status of the last command executed before exit. exit is useful for aborting the shell script if problems occur.

stream manipulation

```
foo | more     # stdout piped into pager
foo |& more     # stdout and stderr piped into pager

bar >foo     # stdout redirected to a file
bar > &foo     # stdout and stderr redirected to a file
```

The first example demonstrates piping the standard out stream, and the standard out and error streams together. In the first case, only standard out would be connected by the pipe to the less command. In the second case both streams are connected to the standard input of the command.

The second example demonstrates the similar combinations for file redirection.